Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Discrete control for ensuring consistency between multiple autonomic managers

Soguy Mak karé Gueye[1*], Noël De Palma[1], Eric Rutten[2], Alain Tchana[1] and Daniel Hagimont[3]

Abstract

The increasing complexity of computer systems has led to the automation of administration functions, in the form of autonomic managers. Today many autonomic managers are available but they mostly address a specific administration aspect which makes necessary their coexistence for a complete autonomic system management. However, coordinating them is necessary for proper and effective global administration. Such coordination can be considered as a problem of synchronization and logical control of managers actions. We therefore investigate the use of reactive models with events and states, and discrete control techniques to solve this problem. This paper presents an application of the latter approach for coordinating autonomic managers addressing resource optimization, in the perspective of green computing. The managers control server provisioning (self-sizing manager) and CPU frequency (Dvfs manager), and the coordination controller controls the managers actions so as to avoid incoherent management decisions. The coordination controller is designed using synchronous programming and Discrete controller synthesis (DCS) which are well-suited for the design of reactive systems. Experimental results are presented to evaluate the efficacy of the approach.

Introduction

Computing systems have become more and more complex and harder to manage manually. Their architecture is mostly distributed involving several hardware and software components operating in a dynamic heterogeneous environment. Manual management of such systems can be time-consuming, expensive and error-prone. Autonomic computing [1] proposes a solution for the management issues consisting in automating the management functions in the form of an autonomic manager. An autonomic manager continuously monitors the managed system so as to detect any change in the system state that is in contrast to its objectives. When it detects such a change, it applies administration operations to lead the system to a state in which its objectives are satisfied. An autonomic manager is generally implemented in a closed loop, which can be inspired by techniques from control theory, continuous as well as discrete. Today many autonomic managers are available but they mostly address a specific administration aspect which makes necessary their coexistence for a complete autonomic system management. However the coexistence of several managers has to be coordinated to avoid incoherent and conflicting management decisions. Implementing manually such a coordination can be tedious, difficult to test and validate, and maintain.

This paper proposes an approach for coordinating multiple autonomic managers based on the discrete control approach. This provides high level programming languages for formal specification of a control system, and tools for automating the verification and validation of properties, and Discrete Controller Synthesis (*DCS*) for the synthesis of the control logic allowing to guarantee the properties, and code generation. Formal descriptions and proofs of correction of the techniques and tools used are presented in [2] and the theories and mathematical models in [3]. The benefit of this approach is the automatic generation of the coordination controller instead of manually programming it in which case it could be complex, tedious and error-prone.

To put our approach into practice, we consider the coordination of two autonomic managers addressing the resource optimization of a system. One addresses the resource optimization within a machine (e.g., *Dynamic Voltage/Frequency Scaling (DVFS)*) while the other

*Correspondence: soguy-mak-kare.gueye@inria.fr
[1]ERODS Team - Bât. IMAG C, 220 rue de la Chimie, 38 400 St Martin d'Hères, France
Full list of author information is available at the end of the article

Springer

© 2013 Gueye et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

addresses the resource optimization within a replication-based system (e.g., *server provisioning (self-sizing)*). These managers act on different management levels, but their actions are complementary to improve resource optimization when coordinated, hence reducing the energy consumption of the managed system.

In the following, we present in section 'Autonomic managers for resource optimization' two autonomic managers, self-sizing and Dvfs. We details in section 'Synchronous programming and discrete controller synthesis' the principles of BZR, a synchronous programming language allowing modelling a system through automata and integrating *DCS* within its compiler. We detail in section 'Discrete control for coordinating self-sizing and Dvfs managers' the design of a coordination controller for the two managers and show how it is integrated within a management system in section 'Implementation'. We show evaluation of the coordinated execution with our approach in section 'Experimentation' and discuss related work in section 'Related work'. Finally, in section 'Conclusion and future work', we conclude the paper and outline directions for future work.

Autonomic managers for resource optimization

This section presents two autonomic managers dealing with resource optimization. Their management decisions consist to reduce the resource allocated to the managed system while preserving good performance. They act at different management levels but their management actions are complementary to improve energy optimization through resource optimization when coordinated.

Server provisioning manager: Self-Sizing

This manager, inspired from [4], addresses the management of the degree of replication of a replicated-based system where each replicated server handles a part of the workload. It dynamically adapts the number of active replicated servers depending on the load of the machines hosting the replicated servers. The load of the machines is measured through the load of their CPU.

As shown in Figure 1, the management decisions of the manager rely on thresholds (maximum threshold and minimum threshold) delimiting the optimal CPU load range. The manager collects the CPU load of the machines hosting the active replicated servers and computes a moving average (**Avg_CPU**). When **Avg_CPU** is higher than the maximum threshold, it considers that the servers (hosts) are overloaded and it adds a new replicated server. When **Avg_CPU** is less than the minimum threshold, it considers that the servers are underloaded, removes a replicated server and turns off the machine that hosts the server.

CPU-frequency manager: Dvfs

This manager, inspired from [5], targets a single machine management. Its role is to dynamically adapt the CPU-frequency of the machine depending on its CPU load.

As shown in Figure 2, the management decisions rely also on thresholds (maximum threshold and minimum threshold) delimiting the optimal CPU load range. When the CPU load of the machine is higher than the maximum threshold, the manager increases the CPU-frequency if the maximum frequency is not reached. It decreases the CPU-frequency when the CPU load of the machine is less than the minimum threshold if the minimum CPU frequency is not reached. This manager runs on the machine it manages. It is implemented either in hardware or software. The one we use is a software implementation and follows the on-demand policy.

Coexistence problem

The coexistence of both self-sizing and Dvfs managers can improve resource optimization. While using the self-sizing manager to optimize the number of active machines allocated to a replication-based system, Dvfs managers can be deployed on each active machine to perform local optimization by adjusting the CPU frequency. However their coexistence has to be coordinated to avoid incoherent management decisions. Indeed when each machine hosting a replicated server is equipped with a Dvfs manager, their CPU-frequency might not be maximal all the time. The Dvfs managers can lower the frequency of the CPU of the machines which makes the latter work slower. In higher frequency a CPU can handle more instructions per time unit than in lower Frequency. A workload which can overload a CPU in lower frequency can possibly be supported by the CPU in higher frequency. So when self-sizing detects an overload in lower CPU frequency, the adding operation it performs can be unnecessary if increasing the CPU frequency of the active machines before the adding operation is sufficient to support the workload. More when an overload occurs in lower frequency and is detected by self-sizing and Dvfs, the adding operation and the CPU increase operations performed by the managers can cause a decrease of the CPU utilization under the minimum threshold leading to removal and CPU decrease operations. When an underload occurs in higher frequency, the removal operations and CPU decrease operations performed by the managers can cause an increase of the CPU utilization over the maximum threshold leading to adding and CPU increase operations.

Coordination strategy

A strategy to achieve an efficient resource optimization and avoid incoherent operations could be to delay as long as possible adding a new replicated server when the machines hosting the active servers are not in their

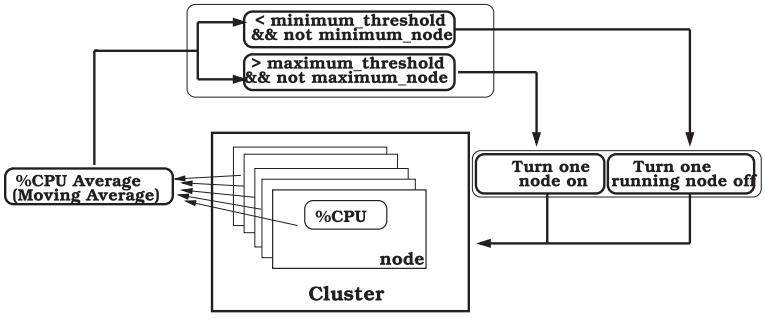

Figure 1 Self-sizing manager. Figure 1 shows the execution scheme of the server provisioning manager. The CPU utilization of the machines hosting the replicated servers are measured and a moving average is computed. If this moving average is higher than the fixed maximum threshold, the manager adds a new server on one unused machine. If the moving average is less than the fixed minimum threshold, the manager removes a server and turns the machine running the stopped server off.

maximum CPU frequency. Indeed the evaluation of an overload by self-sizing is relevant only in higher CPU frequency in which case the CPU frequency can no more be increased. This can be stated as follows:

- **Ignoring overload of the machines hosting the replicated servers** — if these machines are not in their maximum CPU frequency.

This allows to add a new replicated server only when the active ones are in maximum CPU frequency and become overloaded.

Synchronous programming and discrete controller synthesis

Synchronous programming allows to model the dynamics (functional and/or non-functional aspects) of a system as an automaton. A system composed of several sub-systems can be modelled via a composition of automata, where each single automaton models the dynamics of each specific sub-system. Hence, the composition models the state of the system as a whole.

With some controllable transitions in the automata, Discrete Controller Synthesis (DCS) tools [6] can compute a controller that restrains the set of reachable states (i.e., all possible behaviours) to those satisfying a *control objective* (e.g., a coordination policy).

In this Section, we first briefly introduce the basics of the synchronous language Heptagon. We then describe the main features of BZR, that extends Heptagon with a new construct for expressing behavioural contracts [3]. BZR is the language we use for the synthesis of a coordination controller.

Automata and data-flow nodes

The Heptagon language allows the programming of reactive systems by means of mixed synchronous data-flow equations and automata with parallel and hierarchical composition [7]. The basic behaviour is that of the synchronous data-flow languages family [8]: at each reaction step, values of the input flows are used, as well as local and memory values, in order to compute the values of the output flows for that step, and memories for the next step. Inside *nodes* (i.e., block of codes defining an automaton or

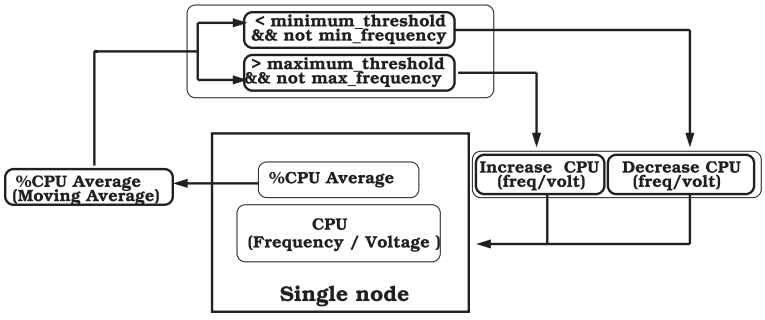

Figure 2 Dvfs Manager. Figure 2 shows the execution scheme of this manager. The CPU utilization of the machine is measured. If the CPU utilization is higher than the fixed maximum threshold, the manager increases the CPU frequency of the machine. If the CPU utilization is less than the fixed minimum threshold, the manager decreases the CPU frequency of the machine.

a composition of automata), these computations are specified as a system of equations defining, for each output and local, the value of the flow in terms of an expression on other flows and memories.

Figure 3 shows a small program in Heptagon. It expresses the control of a `delayable` task that can either be idle, waiting or active. When it is in the initial Idle state, the occurrence of the **true** value on input `r` *requests* the start of the task. Another input `c` (which will be controlled by an external controller) can either allow the activation, or temporarily block the request and make the automaton go to a waiting state (Wait). When in Active, the task can end and go back to the Idle state, upon the notification input `e`. The `delayable` node has two outputs, `a` representing activity of the task, and `s` being emitted on the instant when it becomes active : this latter triggers the concrete start operation in the system's API.

Such automata and data-flow reactive nodes can be composed in parallel and in a hierarchical way. They can be defined and re-used by instantiations of the nodes (see Figure 4 bellow for an illustration, with two instances of Figure 3's node). They run in parallel, defined by synchronous composition (noted " ; "): one global step corresponds to one local step for every equation, i.e., here, for every instance of the `delayable` node.

The compilation of an Heptagon program produces executable code in target languages such as C or Java, in the form of an initialization procedure *reset*, and a *step* procedure implementing the transition function of the resulting automaton. *step* takes incoming values of input flows gathered in the environment, computes the next state on internal variables, and returns values of the output flows. It is called at relevant instants from the infrastructure where the program is used.

Contracts and control in BZR

[3] propose BZR[a] that extends Heptagon with a new construct for expressing behavioural contracts. Its compilation involves discrete controller synthesis [6]. DCS can be described as a formal operation on automata [9]: given an automaton representing all possible behaviours of a system, its variables are partitioned into controllable and uncontrollable variables. For a given control objective (e.g., staying permanently inside a subset of states, considered "good"), the DCS algorithm automatically computes, for each state and value of the uncontrollables, the constraint on controllable variables so that all remaining behaviours satisfy the objective. This constraint is the least necessary, inhibiting the minimum possible behaviours, therefore it is called *maximally permissive*. Formalisms and algorithms are related to model-checking techniques for state space exploration. They are described elsewhere by [6] and [10].

Concretely, the BZR language permits the declaration, using the **with** statement, of controllable variables, the value of which being not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. They are then defined, in the final executable program, by the controller computed by DCS, according to the expression given in the **enforce** statement. A possibility exists, not

$$\texttt{twotasks}(r_1, e_1, r_2, e_2) = a_1, s_1, a_2, s_2$$
enforce not $(a_1$ **and** $a_2)$
with c_1, c_2

$$(a_1, s_1) = \texttt{delayable}(r_1, c_1, e_1)$$
$$(a_2, s_2) = \texttt{delayable}(r_2, c_2, e_2)$$

Figure 4 Mutual exclusion enforced by DCS in BZR. Figure 4 shows the design of a controller in BZR programming language. In this example we compose two instances of the program shown in Figure 3 and enforce a mutual exclusion between two instances.

```
node delayable(r,c,e:bool) returns (a,s:bool)
  let
    automaton
      state Idle
        do a = false ; s = r and c
        until r and c then Active
            | r and not c then Wait
      state Wait
        do a = false ; s = c
        until c then Active
      state Active
        do a = true ; s=false
        until e then Idle
    end
  tel
```

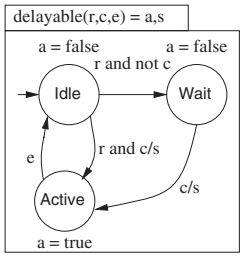

Figure 3 Delayable task in textual and graphical syntax. Figure 3 shows an example of a program with the BZR programming language. In the left hand side, we show the syntax of the language. In the right hand side, we depict the corresponding automaton.

used here, to take into account some knowledge about the environment in an **assume** statement; observers can be used to have objective like: always having a task t_1 between t_2 and t_3". BZR compilation invokes a DCS tool, and inserts the synthesized controller in the generated executable code. The latter has the same structure as above: *reset* and *step* procedures.

Figure 4 shows an example of contract coordinating two instances of the `delayable` node of Figure 3. The `twotasks` node has a **with** part declaring controllable variables c_1 and c_2. The **enforce** part asserts the property to be enforced by DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time: **not** (a_1 **and** a_2). The controllable variables c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the Wait state whenever the other task is in its Active state. Observe that the constraint produced by DCS can have several solutions: the BZR compiler generates deterministic executable code by favouring, for each controllable variable, value **true** over **false**, in the order of declaration in the **with** statement.

Discrete control for coordinating self-sizing and Dvfs managers

This section presents the design of a coordination controller for the self-sizing and Dvfs managers. We first describe the automata modelling the self-sizing and Dvfs managers, then we describe how the coordination controller is designed from these models. The self-sizing is modelled with some control point allowing the control of its operations. In this work, the Dvfs actions are not controlled, we model the global states of the set of Dvfs which are necessary for controlling the self-sizing operations. Indeed the management actions of each Dvfs depend mainly on the load its managed machine receives, which affects the CPU utilization.

Modelling the self-sizing manager

This section presents the automata modelling the self-sizing manager. They represent both the behaviours of the manager (Figure 5) and the control of its operations (Figure 6). The automaton in Figure 5 shows the behaviours of the manager. Initially in the **UpDown** state, When an overload occurs and the upsizing operations are allowed, the manager requests a new node, and goes to the **Adding** state. It awaits in this state until the requested server is available and active. During this period it can no longer perform operations. When **node_added** occurs, the manager returns back to the **UpDown** state or goes the **Down** state if the maximum number of active servers is reached. The **Down** state is left once one node is removed upon an **Underload** event. The **Up** state represents the state in which the degree of replication is minimum and can no longer be decreased. In this case the manager will not perform a downsizing operation regardless the workload (i.e., only upsizing operations can be performed). Table 1 describes the input and output variables of the automaton. The automaton in Figure 6 models the control of the adding operations. Initially in the **Idle** state where adding operations are inhibited, when **c** becomes **true** the automaton goes to the **Active** state allowing to perform adding operations. It stays in this state until **c** becomes **false** and returns back to the **Idle** state. As shown in Table 2, this automaton has one output, i.e., **delay**, which allows upsizing operations when it is **false**. This output feeds the input **delay** in Figure 5. The input and output variables of the automaton are described in Table 2.

Modelling the global states of the set of Dvfs

This section presents the automaton modelling the global states of the set of Dvfs managers presented in Figure 7. Initially in the **Normal** state, the automaton goes to the **Max** state when all Dvfs managers are in their maximum CPU-frequency or to the **Min** state when all Dvfs managers are in their minimum CPU-frequency. It returns back to the **Normal** state when all Dvfs are neither in their maximal frequency nor in the minimal frequency. As shown in Table 3, this automaton has two outputs, **max_freq** being **true** when all local Dvfs reach their maximum frequency and **min_freq** being **true** when all local Dvfs reach their minimum frequency. The input and output variables of the automaton are described in Table 3.

Designing the coordination controller for self-sizing and Dvfs

This section presents the design of the coordination controller for self-sizing and the set of active Dvfs. As shown in Figure 8, the automata modelling the self-sizing and the Dvfs are composed in parallel. The composition of the automata models the uncoordinated coexistence of the managers. The coordination policy is expressed as a contract to be enforced on the latter composition. At compilation DCS automatically generates the control logic capable to attribute value to the controllable variable so as to enforce the coordination policy and restrain the composition to the states satisfying the coordination policy. The composition of the automata and the generated control logic model the coordinated coexistence of the managers.

Contract

To achieve the coordination strategy, we formally define an invariant. The invariant is expressed via the outputs of the automata. It is specified as a contract to be enforced at compilation time.

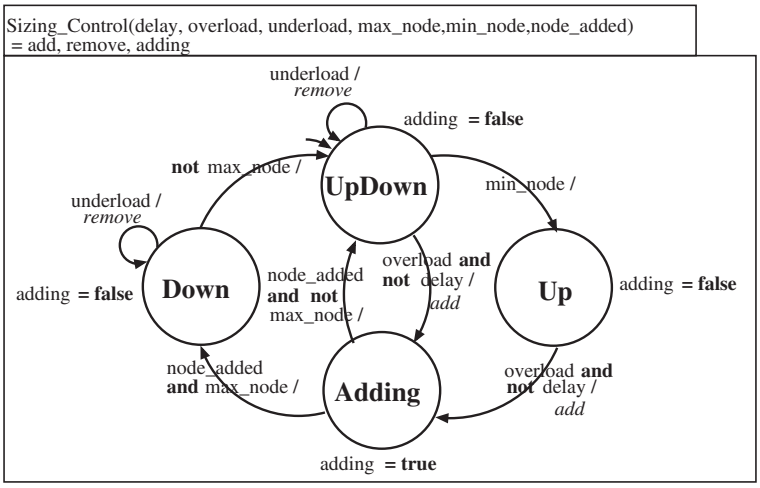

Figure 5 Modelling self-Sizing behaviours. Figure 5 shows the automaton modelling the behaviours of the self-sizing manager.

Coordination policy The strategy consists in preventing the self-sizing manager from adding a new replicated server when the machines hosting the current active servers are not in maximum CPU frequency. This means that the adding operations are inhibited when the Dvfs managers can increase the CPU frequency of their managed machines. To this end, the invariant is defined as follows:

- *invariant* = (`max_freq` **and not** `delay`) **or** (**not** `max_freq` **and** `delay`)

Enforcement

At compilation, the BZR compiler will synthesize a control logic capable of enforcing the coordination policy in the composition by acting on the controllable variable **c** which is an input of the automaton in Figure 6. The composition of the automata and the computed control logic is generated in a target language (i.e., in Java for this work), which will constitute the coordination controller for the managers within the managed system. This coordination policy is very simple but allows to perform the complete experiment including implementation as shown in next section.

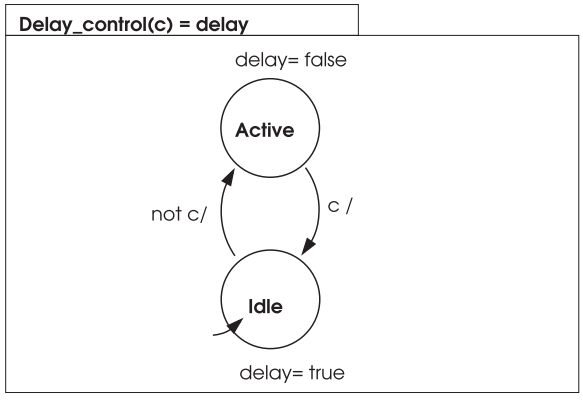

Figure 6 Modelling the control of self-Sizing. Figure 6 models the control of the self-sizing adding operations.

Implementation

This section shows how the generated program from BZR compiler is integrated into the management system. This consists in connecting the inputs of the automata to the corresponding events and the outputs of the automata to the corresponding commands or operations.

Table 1 Variables of the automaton modelling the self-sizing behaviours

Variable	Type	Description
delay	Input	Upsizing operations are suspended.
overload	Input	An overload occurs in the system.
underload	Input	An underload occurs in the system.
min_node	Input	The minimum number of replicas is reached
max_node	Input	The maximum number of replicas is reached.
node_added	Input	The completion of an upsizing operation.
adding	Output	The manager is waiting for the completion of an upsizing operation.
add	Output	The manager is launching an upsizing operation. Its value depends on the value of delay and overload.
remove	Output	The manager is launching the downsizing operation. Its value depends on the value of underload.

Table gives a description of the input and output variables of the automaton modelling the self-sizing behaviours.

Table 2 Variables of the automaton modelling the control of self-sizing

Variable	Type	Description.
c	Input	The upsizing operations must be suspended.
delay	Output	The manager has suspended the launch of upsizing operations.

Table gives a description of the input and output variables of the automaton modelling the control of the self-sizing.

Table 3 Variables of the automaton modelling the global states of the set of Dvfs

Variable	Type	Description
maximum	Input	Corresponds to the conjunction of all *max*.
minimum	Input	Corresponds to the conjunction of all *min*.
max_freq	Output	All Dvfs manager are in the maximum CPU-frequency.
min_freq	Output	All Dvfs manager are in the minimum CPU-frequency.

Table gives a description of the input and output variables of the automaton modelling the global states of the set of Dvfs managers.

Connecting the automata

The automata model an aspect of the dynamics of the self-sizing and Dvfs. Their inputs correspond to the events that trigger transitions between the states in the dynamics (e.g., overload for self-sizing and maximum CPU frequency reached for Dvfs). Their outputs reflect the current state in which the automata are and possibly the operations which should be processed (e.g., add server).

The automaton shown in Figure 7 takes the state of the sef of Dvfs represented by the inputs **maximum** and **minimum**. The input **maximum** (respectively **minimum**) being **true** corresponds to the state where the set of Dvfs reaches the maximum CPU frequency (respectively the minimum CPU frequency). The automaton in Figure 5 models the dynamics of the decision making module of the self-sizing and its control. The input **overload** (resp. **underload**) is the event that triggers **upsizing** (resp. **downsizing**) if **max_node** (resp. **min_node**) is false.**overload** and **underload** are the result of the evaluation of the **CPU_Avg** while **max_node** and **min_node** are the result of the execution of the management operations **upsizing** and **downwsizing**. The triggering of **upsizing** (resp. **downsizing**) is represented by the output **add_node** (resp. **remove_node**) being **true**. The automaton in Figure 6 models the control of the decision making module. It has one input (**i.e., c**) which is controllable. The latter is managed by the synthesized control logic through DCS and the output (**delay**) of the automaton is used to control the value of the input delay in the automaton in Figure 5 in order to control transitions leading to the adding state when necessary.

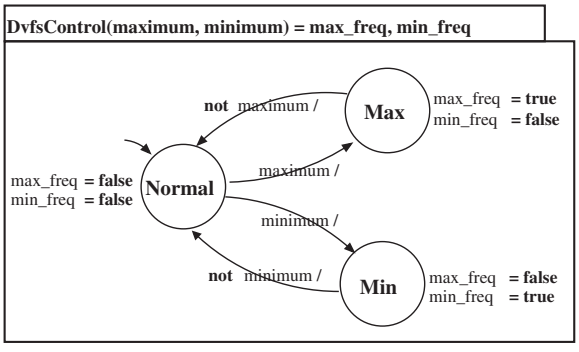

Figure 7 Modelling the global states of the set of Dvfs. Figure 7 models the global states of the set of Dvfs managers.

Integration of the generated code for coordinating managers

The compilation of the BZR program returns a set of Java classes corresponding to the composition of the automata presented above with the computed control logic. A main Java class allows to interact with the program. This class has two methods: *reset* and *step*. The *reset* method allows to initialize the program (i.e., initialize all automata and the generated controller) and the *step* method allows to compute transitions (i.e., transition in the automata). The *step* method takes arguments corresponding to the inputs of the automata and returns outputs corresponding to outputs of the automata. A loop has to be defined to call the *step* with the appropriate inputs and to manage the outputs referring to commands such as preventing upsizing operations of the Self-sizing manager. We implement a loop that receives events from sensors, calls the *step* method with required inputs and transmits the outputs of the *step* method to managers.

Figure 9 represents the architecture of a system in which the coordination controller is integrated. Since the role of this coordination controller is to control which manager should react or not to events, all detected events are first transmitted to the coordination controller. The outputs of the latter are forwarded to the controlled managers i.e in this case the self-Sizing manager. The interface allows interaction between the synchronous program, the sensors and the managers.

Experimentation

In this paper we only focus on the integration of a controller obtained through the Discrete control techniques for the coordination of autonomic managers. The purpose is to show that the latter controller react properly regarding the coordination policy although the system considered is small.

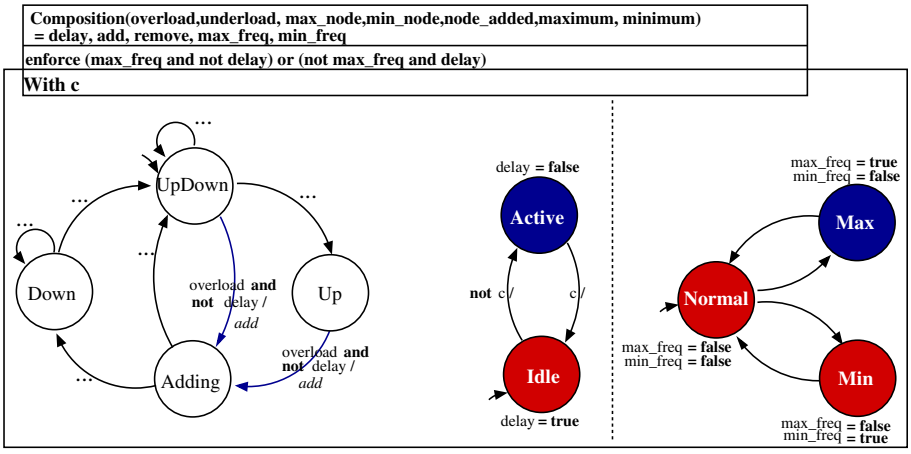

Figure 8 Modelling the coordinated coexistence. Figure 8 models the coordinated coexistence of the self-sizing and Dvfs managers.

The experimental platform, as shown in Figures 10 and 11, consists of a network of three nodes with the same characteristics (CPU, memory, etc.). node 0 hosts the Apache server, the coordination controller and self-sizing. node 1 and node 2 host a Tomcat server. The Apache server acts as a load balancer, it receives all clients requests and distributes the requests to active Tomcat servers for treatment. The self-sizing manager controls the number of active Tomcat servers. Initially, only node 1 is on, self-sizing either turns node 2 on when node 1 is not able to handle all clients requests or off when clients requests can be treated by node 1. node 1 and node 2 have two CPU frequency levels which are 800MHz being their minimum CPU frequency and 1.20GHz being their maximum CPU frequency. The experimental application is CPU bound. We use jmeter[b] to simulate clients sending HTTP requests to the managed system.

In the following we calibrate the thresholds of the managers in order for them to react properly individually at runtime. Then we present executions, to evaluate the behaviours of the synthesized controller.

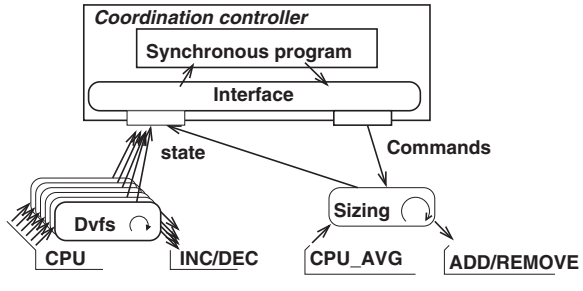

Figure 9 Integration. Figure 9 shows how the generated program can be integrated within a management system.

Calibrating the managers thresholds

This section details how the *Maximum Threshold* and the *Minimum Threshold* for both self-sizing and Dvfs managers are determined. We perform heuristic experimentations to determine these thresholds. For both managers, the *Maximum Threshold* can be statically fixed while the *Minimum Threshold* can be dynamically adapted.

Determining the maximum threshold (T^{max}) for self-sizing and Dvfs

The CPU load is 100 percent means that the CPU is fully utilized. This causes delay on the execution of instructions and the degradation of the performance of the machine. So it is better to consider a maximum threshold less than 100. We choose arbitrary *90 as* T^{max}. At 90 percent, a machine becomes overloaded but it executes instructions in an optimal period of time. This allows to perform operations for monitoring (i.e., CPU load) and reconfiguring (i.e., increase of the CPU frequency) sufficiently fast to avoid performance degradation. The Maximum Threshold (T^{max}) for self-sizing as well as for Dvfs is fixed to 90 percent.

Determining the minimum threshold (T^{min}) for self-Sizing and Dvfs

We use different workloads following the same profile (a ramp-up phase followed by a constant phase), to observe the impact of the management operations of the managers on the CPU utilization. The management operations are performed manually once the workload is constant and stable to evaluate the factor by which the CPU utilization varies. The difference between the workload is the amount of requests injected. This allows to determine if the factor is the same for each workload. This allows to deduct an equation for calculating the *Minimum*

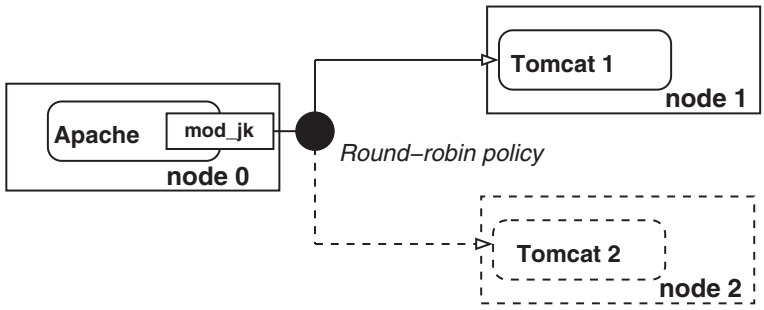

Figure 10 Experimental Platform Architecture: *Managed system*. Figure 10 shows the architecture of the managed system. It is a replicated-based system with replicated Tomcat servers and an Apache server acting as a load balancer.

Threshold based on the *Maximum Threshold*, and the number of active replicated servers for self-sizing.

Minimum threshold (T^{max}) for self-sizing Figure 12 shows experimentations in which adding operations are performed. We start each experimentation with one server. Once the load becomes stable, we add a new replicated server. Since the workload is fairly distributed between the active servers, we expect a decrease of the CPU utilization by half. However the average load measured is always higher than the expected average. Figures 13 shows experimentations in which we perform removal operations. We start each experimentation with two servers. Once the load becomes stable, we remove a replicated server. The load of the remaining server increases but not by factor of two compared to the load before the removal. This means that, for a replicated-based system that can run at most two replicated servers, $T^{min} = T^{max}/2$.

However, in a replicated server-based system there are possibly more than two replicated servers and we want to remove a machine as soon as possible. Removing a server as soon as possible when its workload can be distributed to the remaining servers without overloading them leads to a dynamic estimation of T^{min} depending on the current active servers. This can be expressed as follows:

$$T^{min} + \frac{T^{min}}{(n-1)} < T^{max},$$

where n is the current number of servers.

$$T^{min} < T^{max} * \frac{(n-1)}{n} \longrightarrow T^{min} = [T^{max} * \frac{(n-1)}{n}] - C$$

C is a margin. It denotes the difference between the maximum value of $\boldsymbol{T^{min}}$ and the acceptable value of $\boldsymbol{T^{min}}$ sufficiently high for detecting underload and sufficiently far from $\boldsymbol{T^{max}}$ to avoid oscillations of the CPU utilization between the maximum threshold and the minimum threshold which possibly trigger unnecessary reconfiguration operations. In our experimentations, we consider $C = 0$. In case of two machines, we have: $\boldsymbol{T^{min} = [\frac{T^{max}}{2}] - C}$.

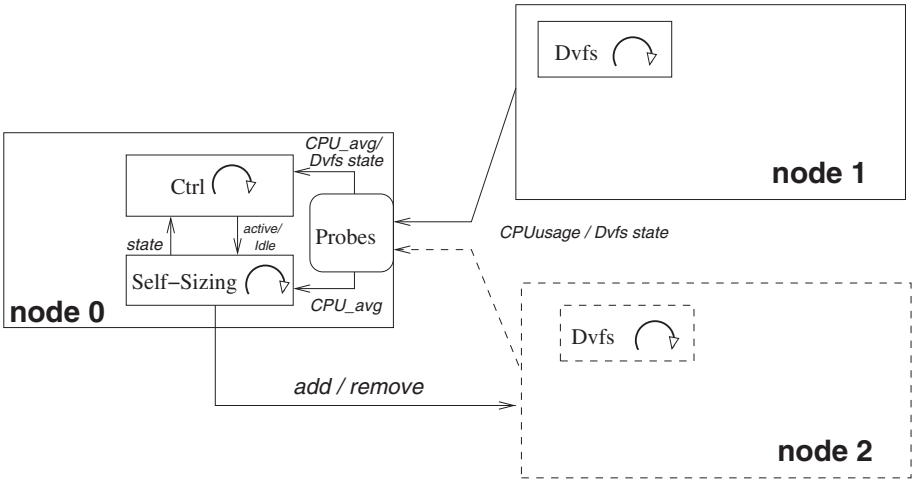

Figure 11 Experimental Platform Architecture: *Administration system*. Figure 11 shows the architecture of the management system.

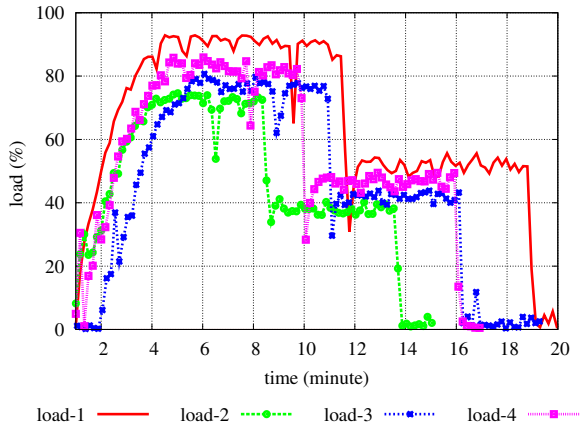

Figure 12 Calibrating self-sizing minimal threshold. Figures 12 shows executions, with different workloads. During these executions, we add a replicated server in order to observe the impact of these operations on the CPU utilization.

As shown in Figures 12, the CPU load does not decrease by half when one server was added and Figures 13, the CPU load does not double when one server was removed. This equation, $T^{min} = [frac T^{max} 2]$, is enough to avoid side effects. So there is no risk of oscillations.

Determining the value of C – To avoid oscillation when the number of servers becomes large a maximum value for T^{min} can be fixed. This avoid T^{min} to be close to T^{max}. In this case the maximum value of T^{min} is used whenever the computed value of T^{min} is higher than the latter.

Minimum threshold (T^{min}) for Dvfs In our platform, the machines hosting replicated server have two CPU frequency levels, 800Mhz and 1.2Ghz. A workload that overloads the machine in frequency 800Mhz could possibly be supported when increasing the frequency up to 1.2Ghz. Theoretically, in frequency 1.2Ghz, the machine performs 1.5 times more instructions than in frequency 800Mhz. This corresponds to the theoretical factor of decrease of the CPU utilization.

Figure 14 shows that the ratio of the maximum frequency to the minimum frequency for different workload profiles. For each workload, the ratio is constant and is less than 1.5. This allows to define the *Minimum Threshold* depending on the *Maximum Threshold* and the ratio of two consecutive frequencies. Indeed, when we put the CPU frequency of machine to a higher frequency than the previous, if the load before the operation was higher or equal to the *Maximum Threshold*, once the operation is done, the load obtained will be higher than (*Maximum Threshold* over 1.5) in our platform. The decrease of the load is higher than the "theoretical" decrease, hence this latter could be used as *Minimum Threshold* since it is reached only if the workload decreases. We can consider the dynamic estimation of the *Minimum Threshold* expressed as follows:

$$T^{min} = T^{max} * \frac{next\ lower\ frequency}{current\ frequency}$$

Coordination controller evaluation

This section presents the evaluation of our approach. We apply our approach for coordinating the self-sizing and Dvfs managers for the management of a replicated-based system. We inject different workload profiles. Each workload profile is defined by two phases, a first phase that consists of a ramp-up load about three minutes then a second phase that consists of a constant the load after the

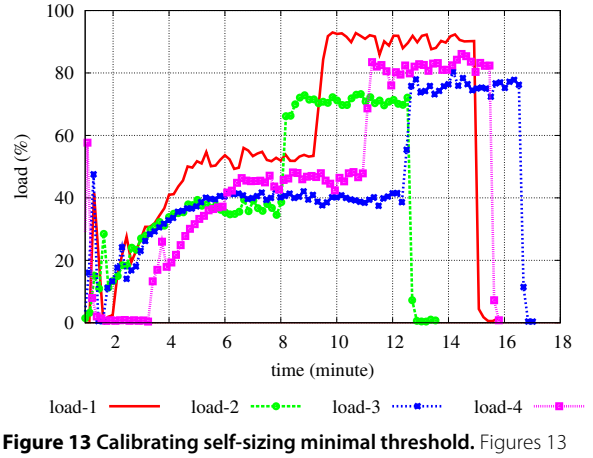

Figure 13 Calibrating self-sizing minimal threshold. Figures 13 shows executions, with different workloads. During these executions, we remove a replicated server in order to observe the impact of these operations on the CPU utilization.

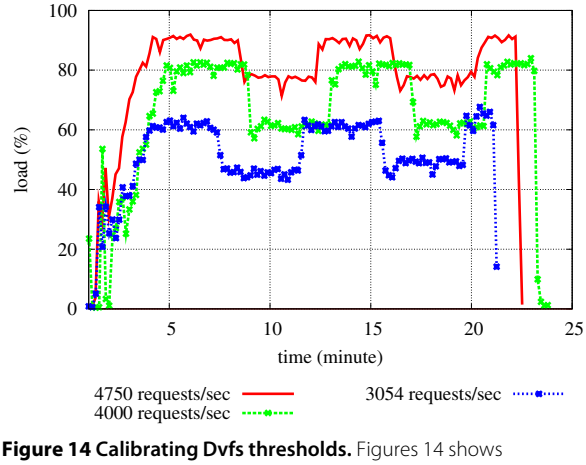

Figure 14 Calibrating Dvfs thresholds. Figures 14 shows executions, with different workloads. During these executions, we changed the CPU frequency on the fly in order to observe the impact of these operations on the CPU utilization.

ramp-up phase. For each workload profile, we performed two executions, one without coordinating managers and another one by coordinating managers execution. At each experimentation, each machine hosting an active Tomcat server starts with its minimum CPU frequency, the local Dvfs adjusts it on the fly. Initially, one Tomcat server is started. The second Tomcat server was either added or removed automatically by the self-sizing manager depending on the workload. The executions takes 20 minutes. After this duration, we stopped sending requests. In this paper, we present three workload profiles *Workload1 (4750 requests/sec), Workload2 (5000 requests/sec) and Workload3 (5542 requests/sec). Workload1* and *Workload2* are supportable by one Tomcat server at max CPU frequency. *Workload3* necessitates two Tomcat servers for treatment.

Without coordination, for *Workload1* (Figures 15) as well as for *Workload2* (Figures 16), the overload detection triggers adding operation and CPU frequency increase operation because the machine hosting the Tomcat server is at minimum CPU capacity. In Figure 15, the overload is detected by self-sizing (*Avg_load*) about 8 minutes after starting sending requests, hence a new server is requested. One minute later Dvfs detects the overload and increase the CPU frequency (*CPUFreq_node1 at 9 min*). Once the new Tomcat server becomes active on node 2 (*about 11 min*), the CPU utilization for both machines hosting the Tomcat servers are around 60% after the Dvfs on node 1 decreases its CPU frequency. We observe the same behaviour in Figure 16. Unlike to uncoordinated executions, in the coordinated executions for *Workload1* (Figures 17) as well as for *Workload2* (Figures 18), only CPU frequency increase is observed. No adding operation is performed when the overload is detected. Once the CPU frequency is executed, the CPU utilization decreases.

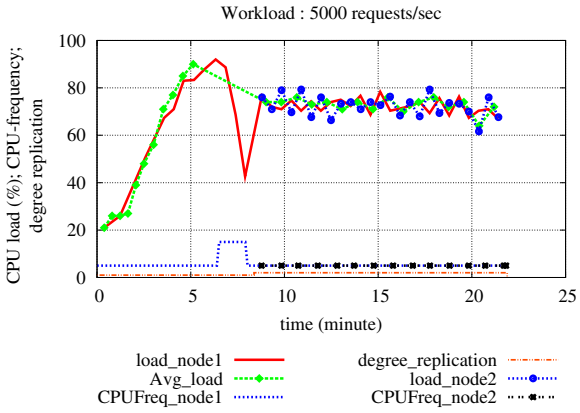

Figure 16 Uncoordinated execution: *Workload2 (5000 requests/sec).* This figure shows an execution in which self-sizing and Dvfs managers are not coordinated.

For *Workload3*, in the uncoordinated execution (Figure 19) as well as in the coordinated execution (Figure 20), a new Tomcat server is added. In contrast to the executions in Figures 19, In Figures 20 the adding operation (*about 6 min*) is performed later after the increase of the CPU frequency on node 1 (*about 4 min*). After increasing the CPU frequency, the overload is not persists (*6 min*) and a new Tomcat server is added since no more CPU-frequency increase operation was possible. The generated coordination controller does not prevent from adding new replicated Tomcat server when it is necessary. The coordination controller is able to ensure the respect of coordination policy. Unlike to the execution without coordination, where undesired behaviours have been observed, we observe that the coordination execution follows the defined policy. Adding operations are

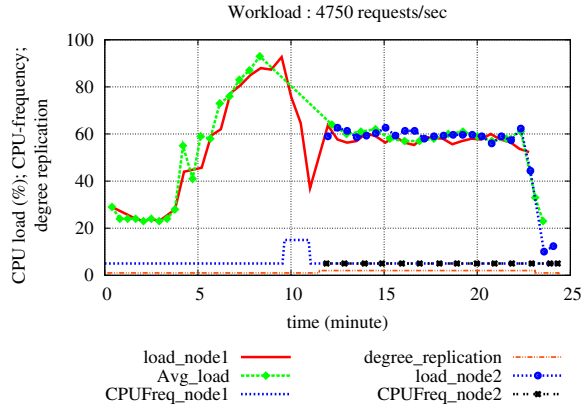

Figure 15 Uncoordinated execution: *Workload1 (4750 requests/sec).* This figure shows an execution in which self-sizing and Dvfs managers are not coordinated.

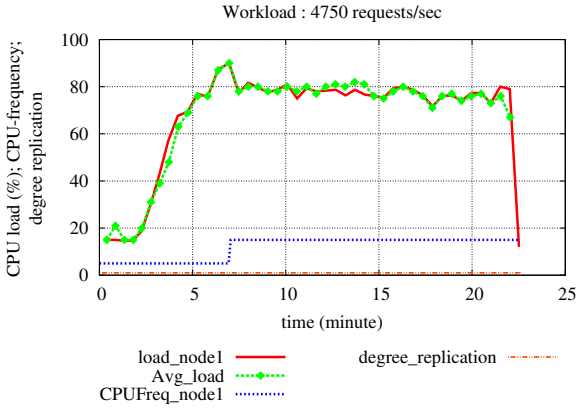

Figure 17 Coordinated execution: *Workload1 (4750 requests/sec).* This figure shows an execution in which self-sizing and Dvfs managers are coordinated.

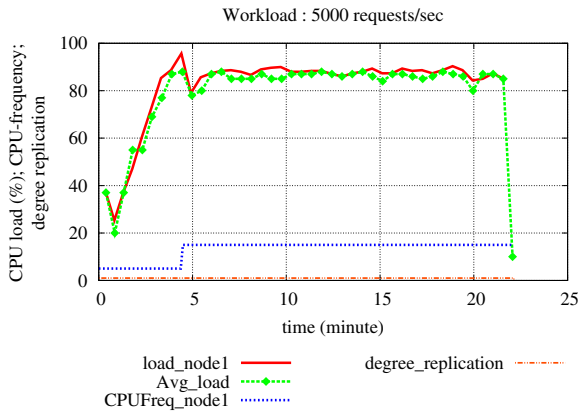

Figure 18 Coordinated execution: *Workload2 (5000 requests/sec)*. This figure shows an execution in which self-sizing and Dvfs managers are coordinated.

Figure 20 Coordinated execution: *Workload3 (5542 requests/sec)*. This figure shows an execution in which self-sizing and Dvfs managers are coordinated.

performed only when all active nodes hosting a Tomcat server are in their maximum CPU frequency.

Related work

Concerning energy control, many works addressed energy management on datacenters. Some of these researches are based on (i) hardware with voltage and frequency control (e.g., DVFS [11]), (ii) resource allocation: Reducing power consumption by reducing the clock frequency of the processor has been widely studied [5,12], Flautner et al. [13] explored a software managed dynamic voltage scaling policy that sets CPU speed on a task basis rather than by time intervals. [14] proposes a power budget guided job scheduling policy that maximizes overall job performance for a given power budget. Many works such as [4,15-18] focused on dynamic resource provisioning in response to dynamic workload changes. These techniques monitor workloads or other SLA (Service Level Agreement) metrics experienced by a server and adjust the instantaneous resources available to the server. Depending on the granularity of the server (single or replicated), the dynamically provisioned resources can be a whole machine in the case of replicated servers. Energy efficiency is achieved using a workload-aware, just-right dynamic provisioning mechanism and the ability to power down subsystems of a host system that are not required.

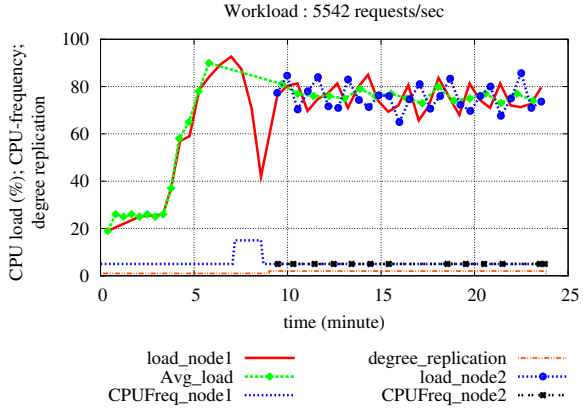

Figure 19 Uncoordinated execution: *Workload3 (5542 requests/sec)*. This figure shows an execution in which self-sizing and Dvfs managers are not coordinated.

While these works are relevant, they did not address the problem of coordinating multiple energy managers. Our work is complementary since it can be used to build a system that includes more that one of the previous approaches. Few works have also investigated manager coordination for energy efficiency. Kumar [19] proposes vManage, a coordination approach that loosely couples platform and virtualization management to improve energy savings and QoS while reducing VM migrations. Kephart [20] addresses the coordination of multiple autonomic managers for power/performance trade-offs based on a utility function in a non-virtualized environment. Nathuji [21] proposes VirtualPower to control the coordination among virtual machines to reduce the power consumption. These works involve coordination between control loops, but these loops are applied to the managed applications. However, these work propose adhoc specific solutions that have to be implemented by hand. If new managers have to be added in the system the whole coordination manager need to be redesigned. Also, the design of the coordination infrastructure becomes complex if the number of co-existing autonomic managers grows. Instead, we propose an approach for coordinating several managers based on control techniques. The latter provide high level programming languages and discrete controller synthesis techniques for the automated synthesis

of the controller capable to ensure the coordination. [22] propose an approach for synchronizing multiple Control-Loops to ensure stability of their behaviours based on a binary linear program. It introduces an Actions Synchronization Module (ASM) that selects, whenever a set of actions needs to be synchronized, the best subset allowed to execute which maximizes a set of QoS metrics. Our approach is similar to the latters since it allows, among a set of actions, a subset to execute. However, contrary to a binary linear program, in our approach the decision is based on the invariants on Control-Loops behaviours. [23] address stability in autonomic networking. It identifies three issues that must be considered to ensure stability which are interactions, conflicts resolution and Time scaling of control-loops. The Game theory approach which provides analytical tools is proposed for studying the efficient collaboration of control-loops. An architectural design is proposed based on the GANA architecture which provides features for structuring control-loops and ensuring their synchronization to achieve stability through Action Synchronization Functions presented in [22]. This approach is based on optimization, typically of QoS metrics, by means of Game theory, whereas our approach proposes an enforcement of logical properties upon states or sequences of actions.

In contrast with [24], which relies on formal specification to derive a formal model that is guaranteed to be equivalent to the requirements, our work can be related to the applications of control theory to autonomic or adaptive computing systems [25]. In particular, Discrete Event Systems in the form of Petri nets models and control have been used for deadlock avoidance problems [26]. Compared to these works, we rely on synchronous programming and discrete controller synthesis. Once an autonomic manager is modelled as automata, inserting this autonomic manager with other pre-existing just require to update the coordination invariants. The new coordination manager is automatically generated from the managers models and the coordination invariants.

In contrast with [27], which addresses the management of datacenters based on thermal awareness with external sensing infrastructure for energy and cooling efficiency, the work, presented in this paper, focuses on coordinating multiple workload-aware managers to ensure an energy efficiency.

Conclusion and future work

One major challenge in system administration is coordinating multiple autonomic managers for correct and coherent system management. In this paper we presented an approach for coordinating multiple autonomic managers in a consistent manner. This approach, based on synchronous programming and Discrete Controller Synthesis, has the advantage of generating by construction the correct controller to enable the coordination of managers.

The advantages of this approach are following: *(1)* High-level of programming, *(2)* Automated generation/synthesis of the controller and *(3)* correctness of the controller, *(4)* that is maximally permissive. The resulting controlled automaton is correct in the sense that the formal technique of DCS has been applied to guarantee, in a form of verification, that it can have only behaviours that satisfy the property to be enforced. It is also maximally permissive in the sense that all behaviours that satisfy the property are kept possible by the controller.

We tested this approach for coordinating two autonomic managers addressing resource optimization: self-Sizing, which manages the degree of replication for a system based on a load balancer scheme, and Dvfs, which manages the level of CPU frequency for a single node. In this case, the coordination policy was to allow self-Sizing to add new node only when all Dvfs modules cannot apply increase operations at all in response to the increasing load the system receives. The experimentations shows that the generated controller ensures a correct coordination with respect of our coordination policy. However, we used thresholds as base for managers decision. These thresholds are not sufficient to capture only overload and underload since there is a probability for a peak of load not to correspond to an overload.

For future work, we plan to improve the model with the use of continuous control to take into account quantitative aspects and avoid oscillations and reduce decision errors. We will improve our use of discrete control by considering more advanced control techniques with cost functions and optimal control. We plan to evaluate this approach for large scale coordination with more complex coordination policies and several managers, combining both self-optimization and self-regulation frequency managers with self-repair manager that heal fail-stop clustered multi-tiers system.

Endnotes

[a] available at http://bzr.inria.fr/

[b] http://jmeter.apache.org/

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
The contributions of the paper are threefold: The use of **synchronous programming** to model the autonomic managers coexistence; The use of **discrete controller synthesis** for the automatic computation of a controller capable to enforce the coordination policy expressed in a declarative way; The **evaluation** of the coordination controller for self-sizing and set of Dvfs for the management of a replication-based system. All authors read and approved the final manuscript.

Acknowledgements
This research is supported by ANR INFRA (ANR-11-INFR 012 11) under a grant for the project ctrl-Green.

Author details
[1]ERODS Team - Bât. IMAG C, 220 rue de la Chimie, 38 400 St Martin d'Hères, France. [2]INRIA Grenoble - Rhône-Alpes, 655, avenue de l'Europe, Montbonnot 38334 St-Ismier cedex, France. [3]IRIT/ENSEEIHT, 2 rue Charles Camichel - BP 7122, 31071 Toulouse cedex 7, France.

Received: 25 February 2013 Accepted: 21 September 2013
Published: 31 October 2013

References

1. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36: 41–50. http://dx.doi.org/10.1109/MC.2003.1160055
2. Marchand H, Gaudin B (2002) Supervisory Control Problems of Hierarchical Finite State Machines In: 41th IEEE Conference on Decision and Control, Las Vegas, USA, pp 1199–1204
3. Delaval G, Marchand H, Rutten É (2010) Contracts for modular discrete controller synthesis In: Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems. LCTES '10, ACM, New York, NY, USA, pp 57–66
4. Chase JS, Anderson DC, Thakar PN, Vahdat AM, Doyle RP (2001) Managing energy and server resources in hosting centers In: Proceedings of the eighteenth ACM symposium on Operating systems principles. SOSP '01, ACM, New York, NY, USA, pp 103–116. http://doi.acm.org/10.1145/502034.502045
5. Weiser M, Welch B, Demers A, Shenker S (1994) Scheduling for reduced CPU energy In: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, OSDI '94. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1267638.1267640
6. Besnard L, Marchand H, Rutten E (2006) The Sigali tool box environment. Workshop on Discrete Event Systems, WODES'06 (Tool Paper). Ann-Arbor, (MI, USA). http://www.irisa.fr/vertecs/Logiciels/sigali.html
7. Colaço JL, Pagano B, Pouzet M (2005) A conservative extension of synchronous data-flow with state machines In: Proceedings of the 5th ACM International Conference on, Conference on Embedded Software. EMSOFT '05, ACM, New York, NY, USA, pp 173–182
8. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proc IEEE 91: 64–83
9. Ramadge P, Wonham W (1987) Supervisory control of a class of discrete event processes. SIAM J. on Control Optimization 25: 206–230
10. Cassandras CG, Lafortune S (2006) Introduction to discrete event systems. Springer-Verlag New, York, Inc., Secaucus, NJ, USA
11. Fox A, Gribble SD, Chawathe Y, Brewer EA, Gauthier P (1997) Cluster-based scalable network services In: Proceedings of the sixteenth ACM symposium on Operating systems principles. SOSP '97, ACM, New York, NY, USA, pp 78–91. http://doi.acm.org/10.1145/268998.266662
12. Govil K, Chan E, Wasserman H (1995) Comparing algorithm for dynamic speed-setting of a low-power CPU In: Proceedings of the 1st annual international conference on Mobile computing and networking. MobiCom '95, ACM, New York, NY, USA, pp 13–25. http://doi.acm.org/10.1145/215530.215546
13. Flautner K, Reinhardt S, Mudge T (2002) Automatic performance setting for dynamic voltage scaling. Wirel Netw 8: 507–520. http://dx.doi.org/10.1023/A:1016546330128
14. Etinski M, Corbalan J, Labarta J, Valero M (2010) Optimizing job performance under a given power constraint in HPC centers In: Proceedings of the International Conference on Green Computing. GREENCOMP '10, IEEE Computer Society, Washington, DC, USA, pp 257–267. http://dx.doi.org/10.1109/GREENCOMP.2010.5598303
15. Lin M, Wierman A, Andrew LLH, Thereska E (2011) Dynamic right-sizing for power-proportional data centers In: Proc. IEEE INFOCOM, Shanghai, China, pp 1098–1106. http://www.caia.swin.edu.au/cv/landrew/pubs/RightSizing.pdf
16. Bouchenak S, De Palma N, Hagimont D, Taton C (2006) Autonomic Management of Clustered Applications In: Cluster Computing, 2006 IEEE International Conference on, pp 1–11
17. Pinheiro E, Bianchini R, Carrera EV, Heath T (2001) Load balancing and unbalancing for power and performance in cluster-based Systems In: Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01). http://research.ac.upc.es/pact01/colp/paper04.pdf
18. Rodero I, Jaramillo J, Quiroz A, Parashar M, Guim F, Poole S (2010) Energy-efficient application-aware online provisioning for virtualized clouds and data centers In: Proceedings of the International Conference on Green Computing. GREENCOMP '10, IEEE Computer Society, Washington, DC, USA, pp 31–45. http://dx.doi.org/10.1109/GREENCOMP.2010.5598283
19. Kumar S, Talwar V, Kumar V, Ranganathan P, Schwan K (2009) vManage: loosely coupled platform and virtualization management in data centers In: Proceedings of the 6th international conference on Autonomic computing. ICAC '09,ACM, New York, NY, USA, pp 127–136. http://doi.acm.org/10.1145/1555228.1555262
20. Das R, Kephart JO, Lefurgy C, Tesauro G, Levine DW, Chan H (2008) Autonomic multi-agent management of power and performance in data centers In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track. AAMAS '08, Richland SC, pp 107–114. http://dl.acm.org/citation.cfm?id=1402795.1402816
21. Nathuji R, Schwan K (2007) VirtualPower: coordinated power management in virtualized enterprise systems In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. SOSP '07, ACM, New York, NY, USA, pp 265–278. http://doi.acm.org/10.1145/1294261.1294287
22. Tcholtchev N, Chaparadza R, Prakash A (2009) Addressing Stability of Control-Loops in the Context of the GANA Architecture: Synchronization of Actions and Policies In: Proceedings of the 4th IFIP TC 6 International Workshop on Self-Organizing Systems, IWSOS '09. Springer-Verlag, Berlin, Heidelberg, pp 262–268. http://dx.doi.org/10.1007/978-3-642-10865-5_28
23. Kastrinogiannis T, Tcholtchev N, Prakash A, Chaparadza R, Kaldanis V, Coskun H, Papavassiliou S (2010) Addressing stability in future autonomic networking. In: Pentikousis K, Calvo RA, García-Arranz M, Papavassiliou S (eds) MONAMI, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, pp 50–61. http://dblp.uni-trier.de/db/conf/monami/monami2010.html#KastrinogiannisTPCKCP10
24. Sterritt R, Hinchey M, Rash J, Truszkowski W, Rouff C, Gracanin D (2005) Towards Formal Specification and Generation of Autonomic Policies In: Embedded and Ubiquitous Computing, pp 1245–1254. http://dx.doi.org/10.1007/11596042_126
25. Hellerstein JL, Diao Y, Parekh S, Tilbury DM (2004) Feedback Control of Computing Systems. John Wiley & Sons
26. Wang Y, Kelly T, Lafortune S (2007) Discrete control for safe execution of IT automation workflows In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. EuroSys '07, ACM, New, York, NY, USA, pp 305–314. http://doi.acm.org/10.1145/1272996.1273028
27. Viswanathan H, Lee E, Pompili D (2011) Self-organizing sensing infrastructure for autonomic management of green datacenters. Ieee Netw 25(4): 34–40. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5958006

doi:10.1186/2192-113X-2-16
Cite this article as: Gueye *et al.*: **Discrete control for ensuring consistency between multiple autonomic managers.** *Journal of Cloud Computing: Advances, Systems and Applications* 2013 **2**:16.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com